An Architecture for the Rapid Development of Robot Behaviors

Jeremy O. Stolarz Paul E. Rybski

CMU-RI-TR-07-16

May 2007

Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

We have developed a mobile platform based on the principlesof modularity and rapid development for studies robotics. Through behavior-based robot control and a unified sensor/actuator architecture we abstract away the details of communicating with the physical hardware, allowing the behavior writer to focus on and more easily develop higher-level controls. Adding to the capabilities of our robot, we have implemented a localization algorithm based on likelihood fields using a laser. Finally, we demonstrate the utility of the FeatureSet architecture in a multi-modal person following behavior which, in addition to fusing laser and vision data to track people, conditions possible tracks using voice feedback from the user.

Contents

1 Introduction

Many robotic systems, especially commercial and industrial ones, are designed with a very specific set of hardware and have software written specifically and only for that robot. While this is generally fine in the domains for which the robots were designed, it is not an optimal solution for more experimental research robotics.

Research centers that investigate robotics often wish to have robots that are more modular in the design of both hardware and software so that they can experiment with different modes of perception and make use of various algorithms. Therefore, it becomes important for the software architecture to easily accommodate new capabilities so that more time can be spent doing the interesting research rather than trying to build a system. To this end, we have created a general robot API called the FeatureSet which provides robot behavior writers with a unified, modular interface to sensors and actuators, abstracting away the nuances of communicating with specific pieces of hardware.

CMAssist, our research group, is interested in studying human-robot interaction (HRI) from the standpoint of mobile robotic assistants/partners. We are primarily focused on issues that involve interactions directly between a human and a robot and have preliminarily focused our attentions on human detection, speech understanding, and task training. As such, our robots must support many modes of sensing and (potentially) actuation, and we use our FeatureSet architecture to facilitate that.

With this system, we have implemented a person tracking and following behavior that uses spoken natural language to correct the robot, should it get off track. The robot can also notify the person it is following that it has lost track and request that he or she come back toward the robot. Such a behavior, itself already a form of HRI, can be employed for even more meaningful interactions. Examples include task training by demonstration and applying semantic labels to the environment.

The two main contributions of this work are:

1. A general robot API that allows for rapid development of higher-level behaviors.

2. A person following behavior that uses speech to improve performance.

The rest of this paper is organized as follows. First, we discuss related work on robot APIs and person following, followed by a description of the robots on which this architecture is deployed. In section 4 we define the requirements for our API and detail its implementation. To demonstrate the versatility of the architecture, sections 5 and 6 give examples of sensors created for the API and a person following behavior, respectively. Some experiments and results performed to test the person following behavior are highlighted in section 7 before we conclude and discuss future work.

2 Related Work

We divide the related work discussion into two sections corresponding to this work's two main areas of contribution. First, we examine other robot APIs; then, we compare our person tracking and following algorithm to others that have been implemented.

2.1 Robot APIs

Before creating a new API, we investigated four existing ones to see if they would fulfill our requirements. ARIA[1] from MobileRobots and Mobility[2] from RWII/iRobot are two commercial APIs we examined. Both are based primarily in C++. One of the main impediments to our adoption of these APIs was that they can not be run in a distributed fashion on multiple computers. Similarly, the open-source TeamBots[3], written mostly in Java, is also restricted to one computer. On the other hand, Player[4], also open-source, is designed with a client/server model over TCP, allowing for distributed computing. However, the components that connect to the sensors and actuators are all required to be written in C++ and only run on *nix operating systems. Our architecture is similar to Player in that it is also based on a client/server model over TCP, but the sensor and actuator components are not restricted to any particular language and can be run on any platform.

[1] http://www.activrobots.com/SOFTWARE/aria.html
[2] http://www.cse.buffalo.edu/ shapiro/Courses/CSE716/MobilityManRev4.pdf
[3] http://www.cs.cmu.edu/t̃rb/TeamBots/
[4] http://playerstage.sourceforge.net/

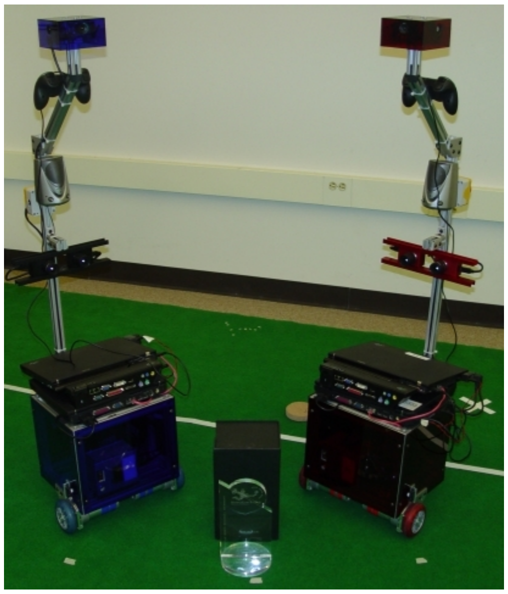


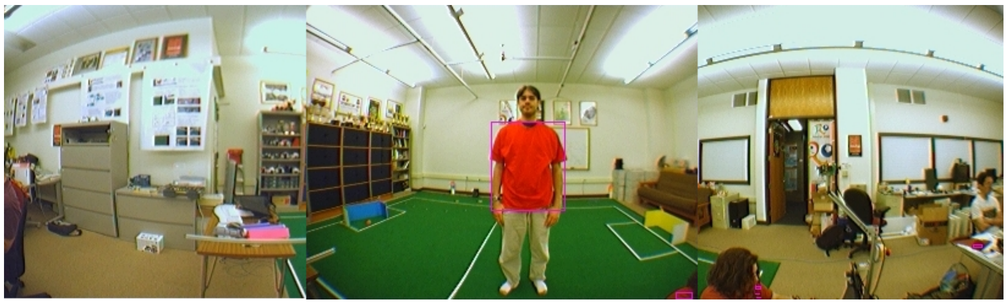
Figure 1: The robots of CMAssist: Erwin and Carmela. The CMAssist team placed 2nd at the 2006 RoboCup@Home competition.

2.2 Person Tracking and Following

Research in tracking has been ongoing for a few decades, but only within the last few years has tracking (and following) of humans been applied to mobile robots. Several person trackers use color blob, face, or contour detection as the primary mode for tracking people such as [1]. However, these methods are prone to failure as the robot moves through areas of different illumination and varying backgrounds. Many other trackers employ laser range scanners to track people, some looking for leg-shaped clusters like [2], others using motion difference [3], and still others comparing readings against a pre-built map [4]. The first method can suffer from clutter of objects that match the profile of legs, the second cannot track stationary people, and the third will not perform well in unknown environments or those that change frequently. It is increasingly common for researchers to combine multiple sensing modalities to overcome the weaknesses of each individual mode. Belloto and Hu [5] and Kleinehagenbrock et al. [6] combine face detection and laser data, while Kobilarov et al. [7] combine color blob detection with laser. A final method of detecting people involves attaching a tracking device to the person [8], but to provide accurate information these devices can be rather bulky, making them overly awkward and intrusive to be worn. Like [7], we combine color blob and laser data, however they only use the vision data to extract features for further identification of people, while we actually fuse the data, allowing the color information to condition tracks, similarly to [5].

Another important part of person tracking (and tracking in general) is data association. In [2] and [5], data association is solved by Nearest Neighbor (NN), the former using Euclidean distance and particle filters and the latter Mahalanobis distance and unscented Kalman filters. NN is a single hypothesis solution to the data association problem. A probabilistic data association filter (PDAF) is employed by [7] and [9] introduce the sample-based joint PDAF (or SJPDAF) for tracking multiple targets. PDAF-class filters can be considered *all-neighbor* [10] association algorithms, generating multiple hypotheses of track/data pairs. Although PDAFs and JPDAFs in particular are proven to be more robust than NN, we chose to use NN for simplicity and because we have assumed our domain of the typical home or office environment will not have much clutter. Additionally, we use a particle filter as in [2] to keep track of possible people.

One area in which our work differs from other person following robots is in the use of the human as a source of information. Similar to [11], in which humans are asked to help the robot when its uncertainty on what to do in various situations is too great, we allow the person being followed to tell the robot if it has begun doing the wrong thing (i.e. began following someone or something else). The robot then uses this information to recondition the tracker and attempt to recover from its error.


(a) Color blob search on color image

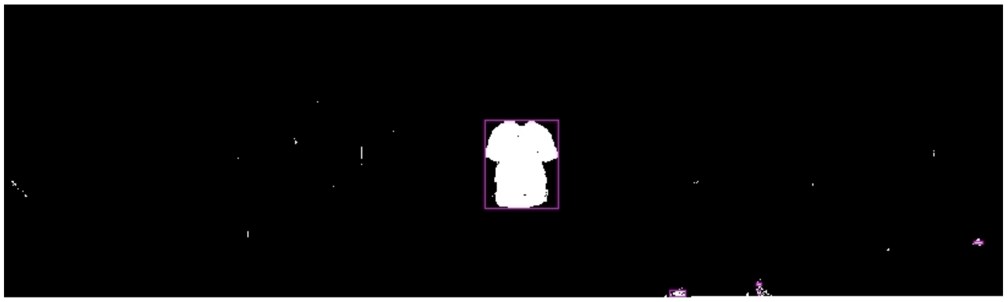

(b) Binary image displaying color blob segments

Figure 2: CAMEO color segmentation.

3 Robotic Platform

The CMAssist robots (shown in Figure 1) were initially based on the ER1 mobile robot platform from Evolution Robotics. However, due to limitations in the motors and structural components, mobility is now provided by a custom set of DC motors, and additional structural hardware was obtained directly from the company 80/20, which manufacturers the aluminum x-bar materials used for the robot's internal structure. The robots have a CAMEO [12] omni-directional camera rig mounted on the top of their sensor mast. They are also equipped with Videre Design's STH-MDCS2-VAR[5] variable-baseline stereo camera rig using lenses with an 80 degree lateral field-of-view. Infrared range sensors are also placed around the base of the robots. A Hokuyo URG-04LX[6] laser range finder (not visible in the picture) is mounted on the base of the robot in front of the laptops. Computational power is provided by two Pentium-M laptops running Linux. A third Pentium laptop running Windows relays voice input captured by a Shure wireless microphone to the NAUTILUS[7] speech processing software. A Logitech joystick can be used to assume manual override control of the robot at any time. An emergency stop button provides further safety by cutting the power to the motors.

Using the CAMEO cameras, the robots are capable of color blob detection [13], used for detection of people (Fig. 2) and other objects of interest, as well as SIFT feature detection [14], used for recognizing landmarks in the environment for improved navigation. Additionally, the stereo cameras provide depth information which is used to create a 2D occupancy grid for obstacle detection.

4 Software Architecture

In deciding upon a software architecture through which behaviors could gain access to and control the various parts of the robot hardware, we had three main objectives: 1) We wished to have a unified, modular interface to the hardware such that behavior designers did not have to worry about the details of communicating with the hardware. 2) We wanted behaviors to be able to run on any robot as long as the robot had to required set of sensors and actuators. 3)

[5]http://www.videredesign.com/sthmdcs2var.htm
[6]http://www.hokuyo-aut.jp/products/urg/urg.htm
[7]http://www.nrl.navy.mil/aic/im4/Nautilus.php

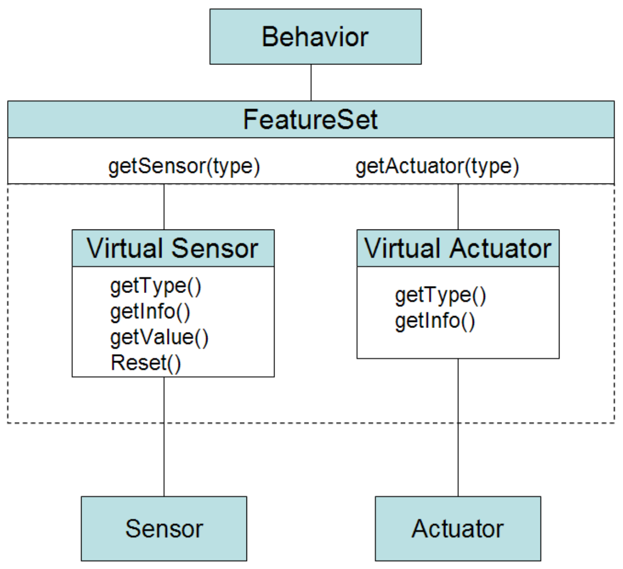


Figure 3: The basic FeatureSet architecture. Behaviors instantiate the FeatureSet, which connects to the robot hardware via its suite of virtual sensors and actuators.

We wished to allow complex combinations of hardware for improved sensing and actuating capabilities while still maintaining the standard interface and transparency of points 1 and 2. The following sections describe in further detail our requirements for the architecture and the details of the implementation of our FeatureSet API.

4.1 Requirements

There were several main requirements that we wanted our API to address. We list them here along with their reasoning in no particular order:

1. As discussed in section 3, multiple laptops are used to communicate with all the hardware and to provide different layers of functionality. Therefore, we needed an API that would allow behaviors to operate transparently on one computer or several computers connected by a network.
2. We wanted communication with hardware to be allowed in any number of languages, so that we could choose the best language for the job on a case by case basis.
3. Since the data coming in from the various sensors is asynchronous, we required it to be timestamped and aggregated appropriately so that behaviors are always running on the newest information possible.
4. We wished for behaviors to be able to run in a cross-platform fashion as some laptops had different operating systems installed on them.
5. Support for a generic set of sensors and actuators was desired as mentioned above.
6. Behaviors should check for the necessary sensor and actuator suite before running and return an error or otherwise act appropriately if no match could be found.

4.2 Implementation

As none of the robot APIs we examined completely fit our needs, we implemented our own, called the FeatureSet. It is loosely based on the behavior/sensor architecture our lab previously developed for use on Sony AIBOs [15].

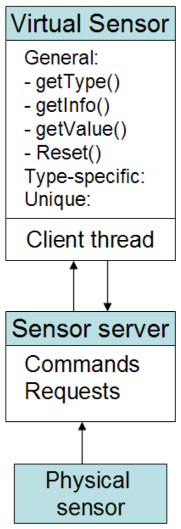


Figure 4: Structure of a virtual sensor in the FeatureSet. Virtual actuators have an analogous setup.

4.2.1 The FeatureSet

The FeatureSet is basically a library of sensors and actuators that a behavior instantiates upon initialization. Behaviors request their desired suite of hardware, and the FeatureSet, via its virtual sensors and actuators, handles the connection to the actual hardware. This basic structure is illustrated in figure 3. Should a given sensor or actuator be unavailable, then the FeatureSet notifies the behavior, so that the behavior can act appropriately.

Like the Player architecture, the FeatureSet is based on a client/server model with connections over TCP in order to allow distributed processing over multiple computers. The servers connect to and handle communication with the physical hardware and can be written in whatever language is deemed appropriate. For example, our robot server, which connects to the motors, is written in Python, while more processor intensive servers like that providing color blob information from the CAMEO cameras are written in C++.

All virtual sensors and actuators in the FeatureSet have a client thread as part of their structure which connects to the relevant server. Communication from between clients and servers is all by character strings to ensure compatibility over different languages and operating systems. The client threads poll the sensor/actuator servers for timestamped data and update threadsafe containers (protected by mutex locks) whenever new information is received. At any time behaviors may request information from the FeatureSet, which returns the most current data from these threadsafe containers. Figure 4 shows the detailed structure of a virtual sensor and its connections. Virtual actuators have an analogous formation.

4.2.2 FeatureSet API

In order for sensors and actuators to have unified and generic interfaces we require that they adhere to the following specifications:

Virtual sensors and actuators must all have these functions defined:

- **getType()** - Returns the type of the sensor or actuator.
- **getInfo()** - Returns type-specific information about the sensor or actuator. Examples include sensor range and units.

Additionally, sensors must define:

- **getValue()** - Returns the sensor readings. Exactly what these are will be type-specific.
- **Reset()** - Reinitializes the sensor.

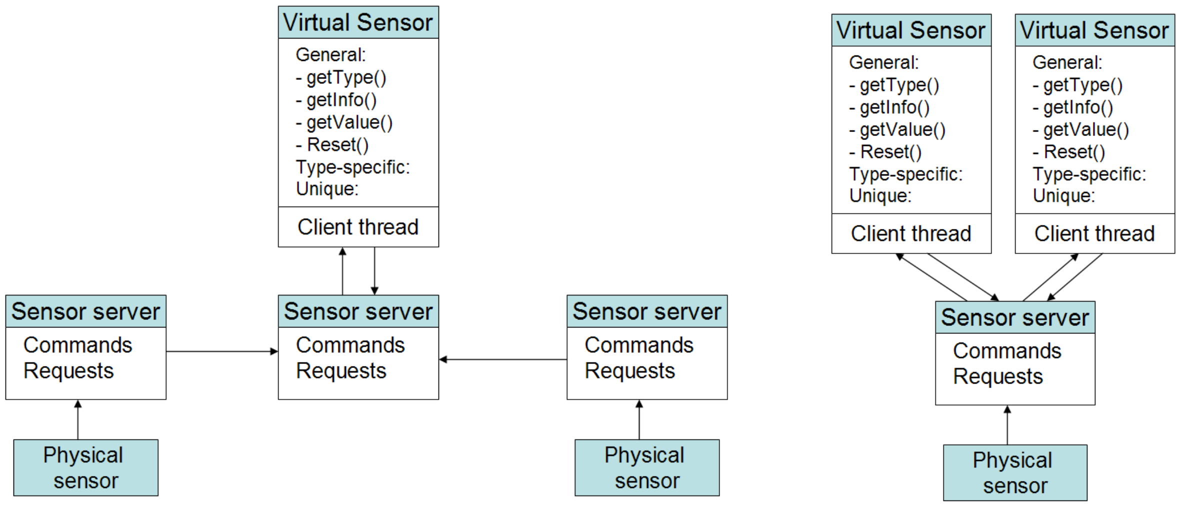
Virtual sensors and actuators can also have a set of type-specific functions defined that would be universal to all sensors and actuators of the same type. For example, actuators of type DIFF_DRIVE (differential drive) must have a **setVelocity()** function which takes in rotational and linear velocities.

Finally, virtual sensors and actuators can have unique functions that add extra functionality for the "power-user" but that are not required for successful usage. These are usually based on the extra capabilities of specific hardware. An example from our robot is the Cepstral speech actuator's **sendCommand()** function which will send a command string to the Cepstral server. A possible use of this function would be to change the tone of the voice.

4.2.3 Complex virtual sensors and actuators

In addition to the basic formulation of virtual sensors and actuators shown in Figure 4, it is possible to make more complex sensor and actuator chains. By having sensor servers communicate with each other, one can create a new sensor server based on the input of multiple physical sensors as shown in figure 5(a). An example of this is the person sensor described in section 6.1 which fuses laser and color blob data to detect "people".

Another possible configuration is shown in Figure 5(b), where a single sensor server is hosting two different virtual sensors. In our design the robot server does this, hosting both a velocity and an odometry sensor, as well as a differential drive actuator.



(a) Virtual sensor accessing a sensor server which combines data from two other sensors

(b) A sensor server hosting more than one virtual sensor

Figure 5: Examples of complex sensor constructs.

5 Laser Sensor

As an example of a FeatureSet sensor, this section presents the laser sensor that we created. At its basic level, the laser sensor provides scan data to the FeatureSet from the laser server, which is connected to the Hokuyo laser. This corresponds to the simple virtual sensor setup in Figure 4. However, we desired to use the laser for improved localization of the robot and decided to use the laser server for this purpose, resulting in the configuration shown in figure 6. The laser server connects to the robot server for odometry information, performs the localization algorithm (see below), then updates the robot server's odometry information to reflect the corrected pose information.

The reason we chose this setup as opposed to having a third sensor for the robot's corrected pose is largely due to backwards compatibility. We already had navigation behaviors that used the odometry sensor for obtaining the robot's pose, and, rather than modify them, we decided to provide this improved information transparently. This exhibits one of the advantages of our FeatureSet architecture, namely the ease with which we can "upgrade" extant behaviors without actually modifying them.

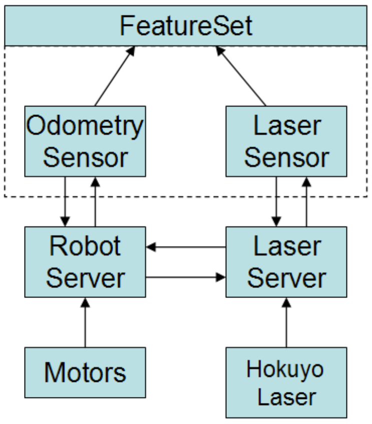


Figure 6: The laser sensor architecture. While the server provides data from the Hokuyo laser to the FeatureSet virtual laser sensor, it also connects to the robot server, updating the pose with localization information.

5.1 Localization algorithm

Originally we wished for our robot to perform SLAM using freely available DPSLAM [16] software. However, we found it ran too slowly to perform in real time. Therefore, we decided to use its off-line mode to build an initial map of the environment from logged laser scan data. The map was built with 5 cm per pixel resolution and then hand-edited for improved accuracy.

The robot localized in the environment using a particle filter. Given an approximate initial position and the pre-built map, the robot was able to track its position using 100 particles (Figure 7(a)). To keep the particles in valid locations (i.e. not inside an obstacle or out of bounds) an obstacle occupancy map was generated from the environment map (Figure 7(b)). Whenever a particle was resampled or moved, a check was made to make sure it was not in an invalid location. If it was, then it was resampled again or its weight was set to zero, depending on if it was being resampled or being updated for movement, respectively. Once particle weights were set to zero they were no longer allowed to have their weights updated by sensor readings until a resample of that particle had occurred. Resamples occurred once the effective sample size of the particles was less than 40% of the original size.

The likelihood field algorithm [17] was used to update the particle weights during sensor updates. The algorithm is as follows:

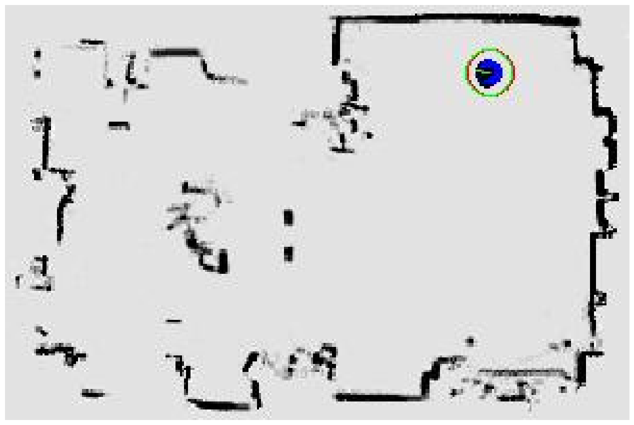
```
Given a robot pose $(x, y, \theta)$, a scan $s$, and a map $m$:
$q = 1$
For all $k$ do
if $s_k \neq s_{max}$
    $x_{s_k} = x + x_{k,sens} cos\theta - y_{k,sens} sin\theta + s_k cos(\theta + \theta_{k,sens})$
    $y_{s_k} = y + y_{k,sens} cos\theta - x_{k,sens} sin\theta + s_k sin(\theta + \theta_{k,sens})$
    $dist = min_{x',y'} \left\{ \sqrt{(x_{s_k} - x')^2 + (y_{s_k} - y')^2} | \langle x', y' \rangle \text{ occupied in } m \right\}$
    $q = q \cdot \left( z_{hit} \cdot \mathbf{prob}(dist, \sigma_{hit}) + \frac{z_{random}}{z_{max}} \right)$
return $q$
```

Where q is the likelihood of the scan s; z_{hit}, z_{random}, and z_{max} are the weights in the probability

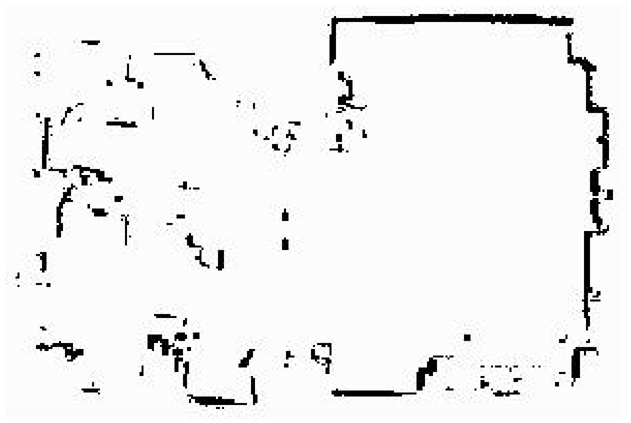
$$p(s_k|x, m) = z_{hit} \cdot p_{hit} + z_{random} \cdot p_{random} + z_{max} \cdot p_{max},$$

and **prob**$(dist, \sigma_{hit})$ computes the probability of $dist$ under a zero-centered Gaussian distribution with standard deviation σ_{hit}.

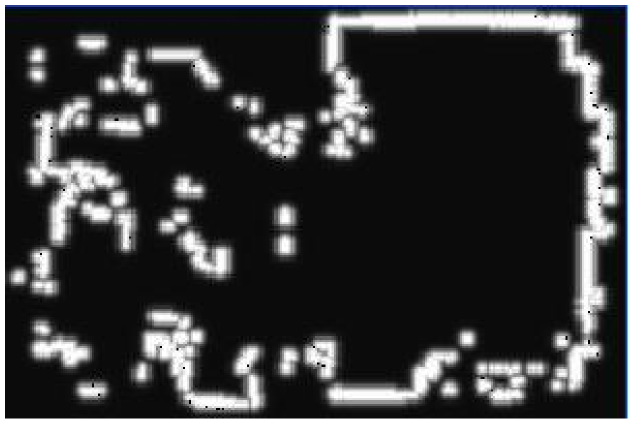
For greater efficiency, the likelihood field (Figure 7(c)) was pre-calculated from the map.



(a) Robot in the environment with current pose in green and particle filter particles shown in blue.



(b) Obstacle map generated from DPSLAM. Black areas represent invalid locations for particles.



(c) Likelihood field. Brighter areas have a higher likelihood of being perceived as an obstacle.

Figure 7: Laser localization

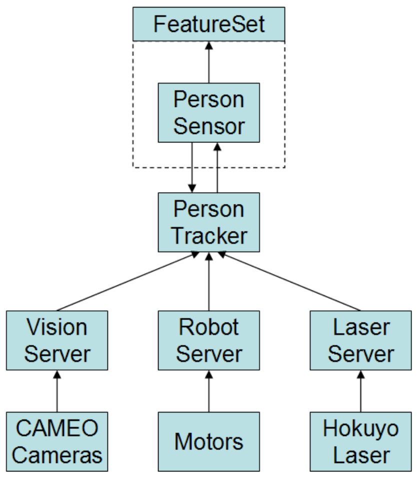


Figure 8: The person sensor architecture. The person tracker takes in data from both the laser and vision servers to identify potential people in the environment. Additionally, it uses the robot server information to update the tracks with the robot's ego-motion.

6 Person Following

The ability for a robot to follow a person is important if the robot is to interact with people. For example, in previous work, we used it to demonstrate a task to the robot that we wished it to learn [18]. Additionally, a robot could add semantic labels to a map if it followed someone around an environment and was told something about the location such as, "this is the living room."

As an example of a behavior written using the FeatureSet architecture, we outline here the person following behavior used by our robots. First, we describe the person tracking sensor, then we discuss the high-level behavior that connects to the FeatureSet and uses the tracker.

6.1 Person Tracker

The person tracker is modeled as a "person sensor" in the FeatureSet as seen in Figure 8. It fuses laser scan data from the laser server and color blob data from the vision server in order to track and return the locations of potential people in the environment. It's structure is similar to that of Figure 5(a).

6.1.1 Leg finding algorithm

The leg finding component of the person tracker parses laser data to identify potential people in the environment based on the clustering of laser points. We have adopted our algorithm from [2]. First, all points that are within 10 cm of each other are considered part of the same segment. Next, each segment is checked to see if it could be a valid person. Segments that are too big (greater than 60 cm wide) or too small (less than 5 cm wide) are discarded. Segments that are too far away (greater than 3.5 m) are also discarded as being unlikely track candidates. Of the segments that have been kept, those greater than 25 cm in width are considered to be a single person, corresponding to the case when someone is standing with their legs together, or if a woman is wearing a dress. Those segments that are less than 25 cm in width are paired with each other as a pair of legs and returned as a single person if they are within 50 cm of each other. Any remaining segments are returned as single segments, corresponding to a person standing sideways from the robot, and thus having only one leg visible. All person candidate locations are returned in (r, θ) coordinates relative to the robot's center, where θ is the midpoint of the cluster and r is the distance to the closest point from the robot. An example of leg detection can be seen in figure 9, where the crosses denote detected candidates.

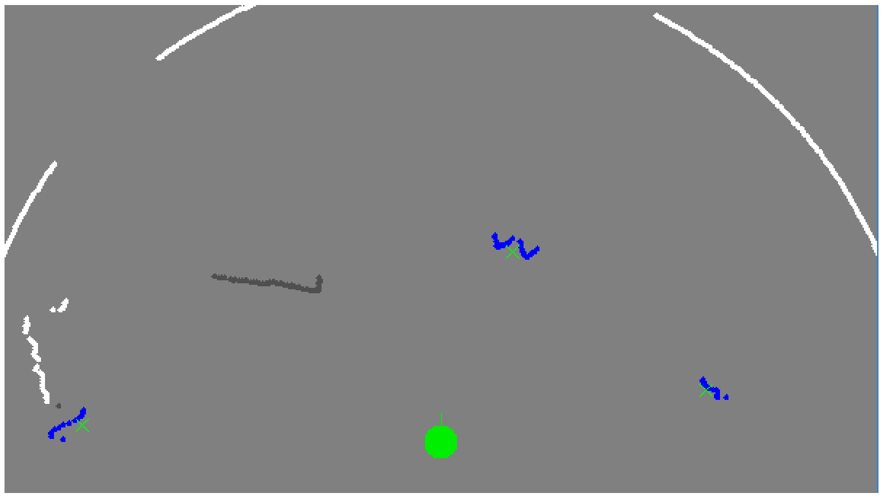


Figure 9: Detecting legs from laser data. Gray segments have been discarded, while blue ones are considered leg candidates. The locations of the candidates returned by the leg finding algorithm are marked by green crosses.

6.1.2 Multiple sensor fusion

Along with the leg data described above, bearing information about color blobs is returned once the data from the vision server is transformed using the cameras' intrinsic parameters. The tracking algorithm use this sensory information to update the estimated location of the person candidates it is tracking.

Each candidate is individually tracked with a particle filter containing 300 particles. We chose to use particle filters rather than Kalman filters as they are able to model non-linear PDFs, which can help deal with the occasional non-linearity of human motion. Tracks exist in (r, θ), with the robot at the origin. Tracks that are updated with laser data have their particles re-weighted according to a 2-D Gaussian in Cartesian coordinates centered at the detected location of the person. Similarly, tracks that are updated with vision data are re-weighted with a 1-D Gaussian centered at the returned bearing θ. Whenever the robot moves all particles (and thus tracks) are moved accordingly to compensate for the robot's motion.

We use a linear motion model to estimate the movements of the tracks during each motion update step of the particle filters. Using the detected locations of the tracks in the previous two timesteps (giving preference to laser data over vision since it provides range and bearing and is more accurate), we calculate a velocity with which to update the tracks' estimated locations. We chose to use a linear motion model rather than the also commonly used Brownian motion model because in our experiments it seemed to provide tracks that were more stable. In addition, they appear to deal better with temporary occlusions.

6.1.3 Data association

The data association algorithm we use is taken primarily from [5]. At a given timestep, the measurements returned from the sensors are $\mathbf{y}^l_m = \begin{bmatrix} r^l_m \\ \theta^l_m \end{bmatrix}$ and $\mathbf{y}^v_n = \theta^v_n$, where $\mathbf{y}^l_m$ contains the bearing and range of the m^{th} leg and $\mathbf{y}^v_n$ is the bearing to the n^{th} color blob. These are then used to update the tracks as follows:

First, the Mahalanobis distance between each track $\mathbf{z}_j$ and each sensor reading $\mathbf{y}_i$ is calculated. The Mahalanobis distance is defined

$$d(\mathbf{y}_i, \mathbf{z}_j) = \sqrt{(\mathbf{y}_i - \mathbf{z}_j)^{\mathrm{T}} \mathbf{\Sigma}_j^{-1} (\mathbf{y}_i - \mathbf{z}_j)},$$

where $\mathbf{\Sigma}_j$ is the weighted covariance of $\mathbf{z}_j$'s particles. Track and observation pairs that have a $d(\mathbf{y}_i, \mathbf{z}_j) \geq \lambda$ are considered unlikely matches and have their distances reset to an arbitrarily large distance. This is based on the notion of a *validation region* [19] constructed by the previous relation. Assuming that d^2 is χ^2 (chi-squared) distributed with n degrees of freedom for an n-dimensional vector, the threshold λ is taken from χ^2 distribution tables. In order to

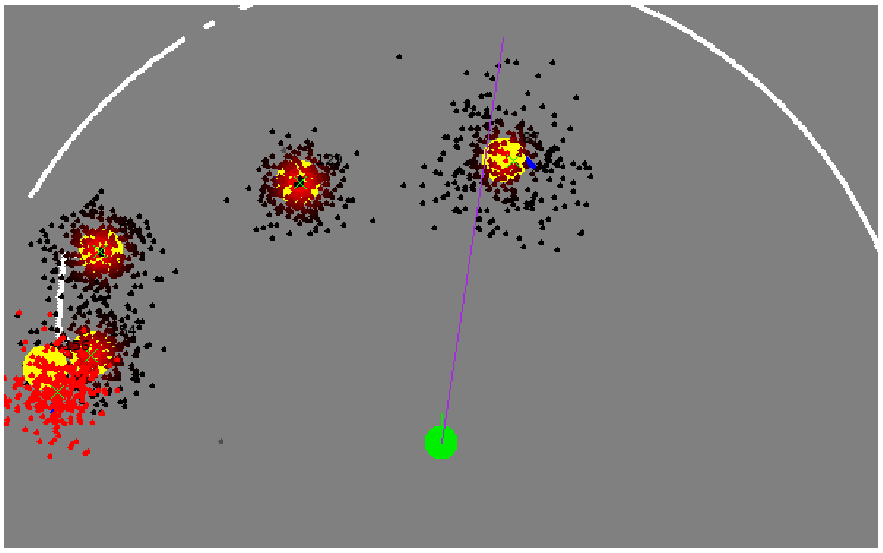


Figure 10: Output from the person tracker. Tracks are marked by the filled yellow circles, marking the weighted centroid of the tracks' particles. Particles range from black to red, red being particles of higher weight. The purple line is the bearing returned from the color sensor, while green crosses are the locations of detected legs. The right-most track corresponds to a person.

have a probability of 0.99 that a reading falls within a track's validation region, we choose $\lambda = 3.03$ for $\mathbf{y}_i^l$ (2-D) and $\lambda = 2.58$ for $\mathbf{y}_i^v$ (1-D).

Next, association matrices are created, where the Mahalanobis distances calculated in the previous step are the individual elements. Since we distinguish between leg and color observations, one matrix corresponds to leg/track associations while the corresponds to color/track associations. The Hungarian Algorithm is then used to compute the globally optimum Nearest Neighbor associations for each matrix.

Observations that have been paired with tracks are used to update those tracks as described in the previous section, while those observations that have not been paired are used to create new tracks. Tracks created from color data are given a range of 1.75 meters (half the maximum tracking distance) with high variance. Tracks that have not been updated by a reading are allowed to persist for 2.5 seconds before being deleted to account for short occlusions.

Additionally, if two tracks come too close together (within 40 cm) one is deleted so as to prevent multiple tracking of the same target. Which track is removed is based on whether the human being followed specified one track as bad (see section 6.2), the number of sensors tracking the track, and the variance of the track's position.

Figure 10 shows an example of visual output from the tracker. Five tracks are currently visible, the right-most of which has both laser and vision sensor readings associated with it, corresponding to a person. The track to the left is of another person that was not wearing a shirt of the proper color and thus cannot be seen by the color blob sensor. The other tracks are false positives due to clutter in the environment.

6.2 Following Behavior

The person following behavior uses information from the person sensor in the FeatureSet to follow people. As seen in figure 11, the behavior begins by instantiating the FeatureSet, requesting access to the person and speech sensors as well as the differential drive actuator. If it cannot load these sensors and actuator, it returns with an error message.

Once it begins, it requests track information from the person sensor. As long as it does not "see" a person about a meter in front of it, the robot will request that the person come within a meter of the robot and make itself visible. A track is "visible" if it has both leg and color data associated with it.

When a person comes close enough, the robot begins to follow him or her, recording the track ID number and striving to keep the person centered and about 1 meter away. On occasion the person tracker will create a new track for a person rather than update the track it had previously been associating with measurements. To account for this, if

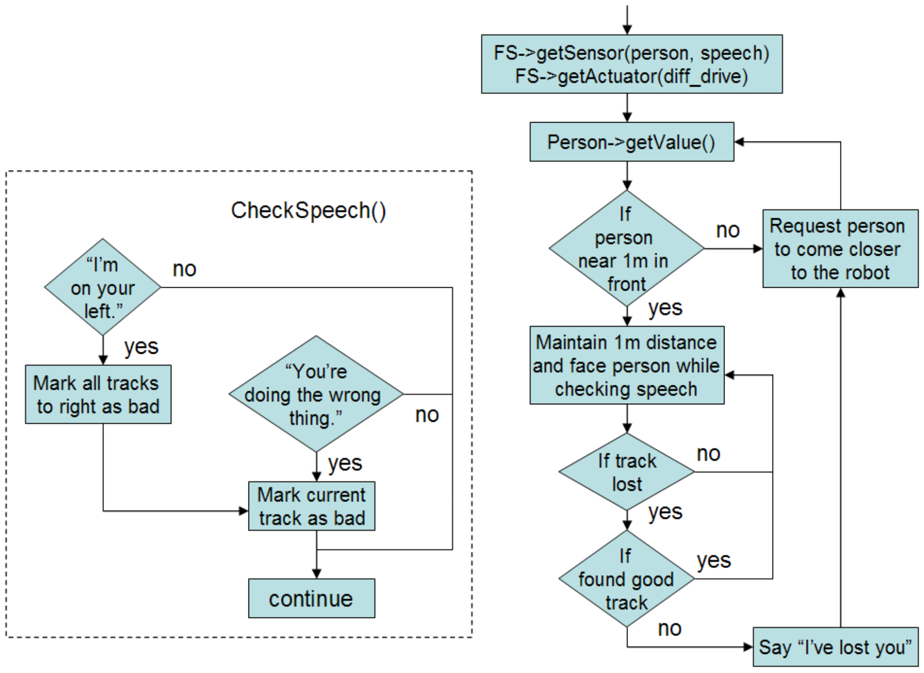

FS->getSensor(person, speech)
FS->getActuator(diff_drive)
Person->getValue()
If person near 1m in front
no
Request person to come closer to the robot
yes
Maintain 1m distance and face person while checking speech
If track lost
no
yes
If found good track
yes
no
Say "I've lost you"
CheckSpeech()
"I'm on your left."
no
yes
Mark all tracks to right as bad
"You're doing the wrong thing."
no
yes
Mark current track as bad
continue

Figure 11: Flow chart of the person following behavior.

the track the behavior is currently following is lost, it will look for another track that matches the one it had previously been following. A track "matches" if it has not been specified as a bad track by the person (see below), has the same sensor agreement as the previous track, and is nearby (within 1.5 meters) the previous track. If another track matches, then its is marked as the track to follow, and the robot continues as before. If it cannot find another matching track, then the robot stops and notifies the user that it has lost track of him or her and requests for them to come back.

While the robot is following someone, it is also continuously listening for an indication from the person that it is no longer following correctly. If the person says something like "You're doing the wrong thing" or "You're following the wrong person," then the robot marks the track it is following as bad and checks for another track as described above. If the person states "I'm on your left," then the robot will mark all tracks to its right as bad. Additionally, it will mark the track it is currently following as bad since it is assumed that the person would only say that if the robot is behaving incorrectly. It again searches for a new track as above. Similarly, if the person says they are to the right of the robot, then it will mark all tracks to its left as bad.

7 Experiments and Results

In order to show how our person tracking and following behavior performs we conducted two tests. The first is meant to show what happens when the person being followed is occluded briefly by another person. The second experiment tests the robot's ability to recover when two people come close enough together to be considered one track. Both of these experiments were run in a typical lab/office setting, where a small amount of clutter could be detected by the robot.

7.1 Test 1

The purpose of this experiment is to show how the robot behaves when someone crosses between the robot and the person it is following. Ten trials were run in which the robot began to follow someone, and shortly thereafter someone else crossed the path of the robot as it tried to follow the first person. As can be seen in table 1, in two cases the person crossing the robot's path caused the tracker to create a new track for the first person, while in only one of those cases did this action cause the robot to completely lose track of the person and stop following him.

Trial	No. lost tracks	Failed
1	0	No
2	0	No
3	2	No
4	0	No
5	0	No
6	1	Yes
7	0	No
8	0	No
9	0	No
10	0	No

Table 1: Results of Test 1

Trial	No. lost tracks	Verbal feedback	Correctly recovered
1	2	No	-
2	4	Yes	Yes
3	5	Yes	Yes
4	3	Yes	Yes
5	0	No	-
6	2	No	-
7	3	No	-
8	2	Yes	Yes
9	0	No	-
10	6	No	-

Table 2: Results of Test 2

This implies that the linear motion model used for maintaining the tracks of people works well when they are briefly occluded by another person.

7.2 Test 2

In this experiment we test how well the robot copes when the person it is following merges with another person before splitting again. In each of ten trials, the robot commenced following someone, while another person stood a couple meters behind and to the side of the first person. The first person would then walk to and stand beside and a bit behind the second person. Once the robot closed the distance to about 1 meter, the person that was being followed stepped away from the other person, walking an additional meter. If the robot failed to follow the first person, he then told the robot it was following the wrong person.

As shown in table 2, in all but two cases, the original track of the person being followed was lost by the tracker a number of times. This is likely due to the tracking algorithm eliminating tracks when they are within 40 cm of each other. In six trials, the robot successfully continued following the first person after he stepped away from the second person. Except for the two cases in which the tracker did not lose track of the first person, it is unclear whether this was due to random luck or not. In the four cases where the robot did not follow the right person, after being told it was following the wrong person, it correctly recovered and began following the original person (though in three of those cases the original person was out of range and the robot requested that he come back toward the robot).

These results imply that the robot's response to being told it is following the wrong person result in the robot properly ignoring the false track and recovering when the correct candidate is in range.

8 Conclusion

We have described our FeatureSet API which provides a modular, unified architecture to access robot sensors and actuators. To show the versatility of our architecture, we implemented a complex virtual laser sensor that also performed passive localization, transparently updating the robot's pose. In addition, we created a person following behavior that fuses laser and vision data and uses voice feedback from the human being followed to aid it when lost. Preliminary experimental results indicate that our approach successfully allows the robot to follow someone even if they are occluded for a short period, and that if the robot receives negative feedback from the human it can properly recover from its error.

The abstraction of the FeatureSet facilitates behavior development in three main ways. First, the behavior writer does not need to worry about how to communicate with given sensor or actuator as they all follow the same convention. Second, the same behavior can be run on any robot as long as it has the requested suite of sensors and actuators. Finally, virtual sensors and actuators can transparently represent complex combinations of physical hardware as shown by the laser and person sensors above.

9 Future Work

There are several directions in which CMAssist would like to expand. To improve sensing capabilities we would like to switch to a more robust color segmentation technique that models colors as probability distributions that can be learned and that can adapt to changing illumination such as in [1] and [20]. In the domain of person following, we hope to integrate the following behavior with navigation for improved tracking and following of a human when he or she moves around an obstacle. Additionally, we have begun joint work with members of both the School of Design and the Human-Computer Interaction Institute here at Carnegie Mellon in designing and conducting studies with a "snackbot" that will visit offices of faculty and students to sell and deliver snacks.

10 Acknowledgements

This work is part of the CMAssist effort of the CORAL research group. Other CMAssist and CORAL members that have contributed to aspects of this work include Kevin Yoon, Alex Grubb, Stefan Zickler and Chris Broglie. We would also like to thank the Naval Research Labs for developing the NAUTILUS natural language processing system and for helping us understand how to use it.

References

[1] Schlegel, C., Illmann, J., Jaberg, K., Schuster, M., Wörz, R.: Vision based person tracking with a mobile robot. In: Proceedings of the Ninth British Machine Vision Conference (BMVC), Southampton, UK (1998) 418–427

[2] Gockley, R., Forlizzi, J., Simmons, R.: Natural person-following behavior for social robots. In: Proceedings of Human–Robot Interaction, Arlington, VA (March 2007) 17–24

[3] Lindström, M., Eklundh, J.O.: Detecting and tracking moving objects from a mobile platform using a laser range scanner. In: Proceedings of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), Maui, Hawaii (November 2001)

[4] Montemerlo, M., Thrun, S., Whittaker, W.: Conditional particle filters for simultaneous mobile robot localization and people-tracking. In: Proceedings of the IEEE Int. Conference on Robotics and Automation (ICRA), Washington, DC (May 2002) 695–701

[5] Bellotto, N., Hu, H.: Vision and laser data fusion for tracking people with a mobile robot. In: Proceedings of the IEEE International Conference on Robotics and Biomimetics (ROBIO), Kunming, China (December 2006)

[6] Kleinehagenbrock, M., Lang, S., Fritsch, J., Lömker, F., Fink, G.A., Sagerer, G.: Person tracking with a mobile robot based on multi-modal anchoring. In: Proceedings of the 2002 IEEE Int. Workshop on Robot and Human Interactive Communication, Berlin, Germany (Sept 2002) 423–429

[7] Kobilarov, M., Sukhatme, G., Hyams, J., Batavia, P.: People tracking and following with mobile robot using an omnidirectional camera and a laser. In: Proceedings of the IEEE International Conference on Robotics and Automation, Orlando, Florida (May 2006) 557–562

[8] Gigliotta, O., Caretti, M., Shokur, S., Nolfi, S.: Toward a person-follower robot. In: Proceedings of the Second RoboCare Workshop. (2005)

[9] Schulz, D., Burgard, W., Fox, D., Cremers, A.B.: People tracking with a mobile robot using sample-based joint probabilistic association filters. International Journal of Robotics Research **22**(2) (2003)

[10] Blackman, S., Popoli, R.: Design and Analysis of Modern Tracking Systems. Artech House, Boston, MA (1999)

[11] Fong, T., Thorpe, C., Baur, C.: Collaboration, dialogue, and human-robot interaction. In: Proceedings of the 10th Intl. Symposium of Robotics Research, Lorne, Australia (November 2001)

[12] Rybski, P.E., de la Torre, F., Patil, R., Vallespi, C., Veloso, M.M., Browning, B.: Cameo: The camera assisted meeting event observer. In: Proceedings of the 2004 IEEE International Conference on Robotics and Automation, New Orleans (April 2004)

[13] Bruce, J., Veloso, M.: Fast and accurate vision-based pattern detection and identification. In: Proceedings of the 2003 IEEE International Conference on Robotics and Automation, Taiwan (May 2003)

[14] Lowe, D.: Object recognition from local scale-invariant features. In: Proceedings of the Seventh IEEE International Conference on Computer Vision, Corfu, Greece (September 1999) 1150–7

[15] Lenser, S., Bruce, J., Veloso, M.: CMPack: A complete software system for autonomous legged soccer robots. In: Proceedings of the Fifth International Conference on Autonomous Agents. (May 2001)

[16] Eliazar, A.I., Parr, R.: Dp-slam 2.0. In: Proceedings of the IEEE International Conference on Robotics and Automation, New Orleans, LA, USA (April 2004)

[17] Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics. The MIT Press, Cambridge, MA (2005)

[18] Rybski, P., Yoon, K., Stolarz, J., Veloso, M.: Interactive robot task training through dialog and demonstration. In: Proceedings of the 2007 Conference on Human-Robot Interaction, Washington DC, USA (March 2007)

[19] Bar-Shalom, Y., Fortmann, T.E.: Tracking and Data Association. Academic Press, Inc., San Diego, CA, USA (1988)

[20] Sridharan, M., Stone, P.: Autonomous planned color learning on a mobile robot without labeled data. In: Proceedings of the IEEE Ninth Int. Conf. on Control, Automation, Robotics, and Vision (ICARCV), Singapore (December 2006)

11 Appendix

This appendix explains how to use and interpret the data from the laser server and person tracker as well as the person following behavior.

11.1 Laser Server

The laser server can be found in `er1/laser`. Most of the information here can also be found in the header of `laser_server.cpp`. Laser_server is written to communicate with the Hokuyo laser on the CMAssist robots.

11.1.1 Execution

To execute the laser server, first make sure the laser is on and that er1server is running. Then run `laser_server` followed by any of these options:

- `-p <port_number>`: specifies the TCP port that laser_server listens to.
- `-l <log_filename>`: specifies the filename to save laser data to. This data is saved in a format that can be used by the DPSLAM software to build a map offline.
- `-v`: shows Hokuyo laser visualization.
- `-z`: turns on debug output.
- `-e <host>`: specifies er1 server host.
- `-f <port>`: specifies er1 server port.
- `-c <PF_cfg_filename>`: specifies the particle filter configuration filename. See section 11.1.2.
- `-m <map_filename>`: specifies the environment map filename. By setting this flag the laser server will localize the robot in the map. The environment map should be a grayscale image, with black marking obstacles/out of bounds areas and white free space.
- `-o <offline_logfile>`: laser_server runs offline from the data in the specified log file. This log file should be in the same format as the DPSLAM log file.
- `-g`: turn on particle filter graphics.
- `-d`: visualize laser beams from particles (`-g` required).
- `-s <frequency>`: save particle filter graphics to file with given frequency in ms (`-g` required).
- `-h`: prints the usage menu and quits.

NOTE: `-l` and `-m` are mutually exclusive options. You cannot log data for DPSLAM and do localization at the same time.

To quit the program, you can either press `Esc` or `q`.

11.1.2 Particle filter configuration file

The optional particle filter configuration file can provides the laser server particle filter with values for the following parameters, overriding the defaults. Each parameter should be on its own line. Additionally, any line that begins with # is considered a comment and ignored by the parser:

- `MaxNumParticles <val>`: Specifies the number of particles in the particle filter.
- `ResampleThresh <val>`: Specifies the particle resampling threshold.

- `MotionVariance <lin_vel_sigma> <rot_vel_sigma>`: Specifies how much noise should be used for the robot's noise model during update for motion. **NOTE:** `lin_vel_sigma` is in m/sec and `rot_vel_sigma` is in deg/sec.
- `PoseResampleVariance <x_sigma> <y_sigma> <theta_sigma>`: Specifies the variance that should be applied to a particle's pose when it is resampled. **NOTE:** `theta_sigma` is in degrees and `x_sigma` and `y_sigma` are in meters.
- `BEAM_ANGULAR_SAMPLE_RATE <degrees>`: Specifies the angle interval at which the PF sensor update should take scan range data. In other words, it only uses the scan point at every `<degree>` degrees.

The following parameters can also be read from the configuration file and are relevant to generating the Likelihood field:

- `Z_HIT <val>`: Specifies the weighting to give to the probability of getting a reading from the laser.
- `Z_RANDOM <val>`: Specifies the weighting to give to the probability of random noise.
- `Z_MAX <val>`: Specifies the weighting to give to the probability of getting a max range reading.
- `SIGMA_HIT <val>`:Specifies the variance for sensor readings when a hit occurs

The following two parameters are used when you want to solve more of a "tracking" or "local" kidnapped robot problem. `x` and `y` are in meters, and `theta` is in degrees:

- `InitialPose <x> <y> <theta>`: Specifies the initial pose of the robot. This is set in both er1server and in the distribution of particles. Particles are distributed around this point according to `InitialPoseVariance`.
- `InitialPoseVariance <x_sigma> <y_sigma> <theta_sigma>`: Specifies the variance with which the particles should be sampled around the robot's start position, specified by `InitialPose`.

11.1.3 Laser server commands

To use the laser server, a client can send the following requests, and will receive the corresponding responses (NOTE: case counts!):

- `getPose`: returns the estimated pose of the robot
  Returns: `pose x y theta timestamp`,
  where `x` and `y` are in meters, `theta` in radians, and `timestamp` in milliseconds
- `getLaserScan`: returns the most recent laser scan in several packets, each value a distance in meters
  1st packet - `scan num_packets timestamp s1 s2 s3 s4 ...`
  subsequently - `s_x s_x+1 s_x+2 ...`,
  where `num_packets` is two bytes long and specifies how many packets (including the first) are being sent, and each value `s_x` a distance in meters. The `timestamp` value is a float. The client should read each packet and concatenate it to the previous ones before parsing as it is possible for a scan value to be split over packets. Each packet has a maximum length of 1024 bytes, including the null byte.
- `getAngularRes`: returns the angular resolution of the laser in radians
  Returns: `angularRes val`
- `getScanAngle`: returns the scan angle of the laser in radians
  Returns: `scanAngle val`
- `getRange`: returns the maximum range of the laser sensor in meters
  Returns: `range val`
- `getNegAngThresh`: returns the negative angle threshhold of "bad data" due to occlusions caused by the robot wheels in radians
  Returns: `negAngThresh val`

- `getPosAngThresh`: returns the positive angle threshhold of "bad data" due to occlusions caused by the robot wheels in radians
  Returns: `posAngThresh val`
- `streamScan <0/1>`: tells the server to start or stop streaming scan data. 0 is off, 1 is on. Since we have no way to keep track of individual clients with our socket code, once streaming has been turned on, it will only turn off if only one client is connected at the time of the request. That way if multiple clients requested it to be on they won't get messed up. All clients should therefore be able to handle scan data coming in even if they don't ask for it (that goes for all data, really). No ack is sent be the server when it receives this command. Also, as good form, all clients that request streaming should "turn off" streaming before disconnecting.

If an unknown/unrecognized request is sent to the server, it will respond with `unknown`.

11.1.4 Particle filter graphics

If particle filter graphics are enabled (see section 11.1.1), then laser_server will display something similar to figure 7(a). The current pose estimate as returned by the particle filter is in red, while the pose from the robot server is in green. Particles will vary in color from black to blue depending on their weights. The higher the weight, the brighter the blue.

11.2 Person Tracker

The person tracker sensor can be found in `er1/person_tracker`. Much of the information here can be found in the header of `person_tracker.cpp`.

11.2.1 Execution

Since the person tracker relies on vision and laser data as well as pose information, before it can be executed `laser_server`, `yuv_multicolor_server2`, and `er1server` must all be running. When ready, execute `person_tracker` with any of the following options:

- `-p <port_number>`: specifies the TCP port that the person tracker server listens on.
- `-e <host>`: specifies the er1 server host.
- `-f <port>`: specifies the er1 server port.
- `-l <host>`: specifies the laser server host.
- `-m <port>`: specifies the laser server port.
- `-v <host>`: specifies the vision server host.
- `-g`: enables person tracker visualization.
- `-s <frequency>`: save graphics with given frequency in ms (requires `-g`)
- `-t <filename>`: log graphics to given file in text form (requires `-g`). Files generated with this option will grow in size quickly. The idea behind this was to then parse this file to create graphics with the DrawView functions, however, said parser has not been written. This was an attempt to solve the drop in tracking performance when saving the graphics to file with the `-s` option.
- `-k`: enables a debug mode in which significant person tracking steps are "stepped through" via keyboard control. `Space` proceeds to the next step. This is useful for getting a grasp on what is occurring at each phase of person tracking.
- `-c <filename>`: specifies the Tsai camera calibration file location. This file is necessary so that ColorBlob data can be transformed into bearing information. Currently, a copy of this file can be found in `person_tracker/tsai_cam_cal.txt`.
- `-h`: prints the usage menu and quits.

11.2.2 Person tracker server information

The person tracker server operates in much the same manner as yuv_multicolor_server. Once a client is connected to the server, it will start streaming information about detected people. Below is the format of the packets from person tracker:
`<timestamp> <numTracks> <ID> <r> <theta> <numSensors> <bad_flag> ...,`
where `numTracks` is the number of people detected. For each track, `ID` is the unique track ID, `r` is the distance to the person in meters, `theta` is the bearing of the track in radians (positive is to the robot's left, negative to its right), `numSensors` is the number of sensors that agree on the track, and `bad_flag` states whether this track was previously marked as a bad track by the person following behavior.

Following are commands the server recognizes from clients:

- `bad <num> <ID> ...`: tells the server which tracks are "bad," so it can set the flag appropriately. `<num>` is the number of tracks being marked bad, and then each track ID should be sent separated by spaces
- `curr <ID>`: tells the server which track is currently being followed. This is currently just for display purposes, though it could probably be used to help in judging which tracks to eliminate as well.

11.2.3 Person tracker visualization

If the visualization for the person tracker is enabled (see section 11.2.1), then person_tracker will display something similar to figure 10. Each track will be marked by a big colored circle, where the center of the circle is located at the robust mean of the particles and the width of the circle corresponds to the robust mean radius threshold. If the circle is green, it is marked as being followed by the person following behavior, while if it is orange, it has been marked as a "bad" track by the behavior. Yellow tracks have no markings from the behavior (default). The particles of each track track are also displayed ranging in color from red to black, where red particles have the highest weights.

In addition to the weights, data from the sensors is also displayed. A purple line extending from the robot (in light green) marks the bearing of a detected color blob. Laser data is displayed in a few different colors depending on how it has been interpreted by the leg detection algorithm. White points are ignored; gray points are part of a contiguous segment that has been rejected as too big, small, or far away to be legs; while blue points correspond to segments that are considered legs. The green crosses mark the center point of each "leg", which is the information the tracker uses for determining person locations.

11.3 Follow person behavior

The person following behavior that uses the person sensor can be found at `er1/behaviors/follow_person.py`. Along with the standard er1 sensors and actuators it instantiates the FeatureSet asking for the person and speech sensors and the Cepstral actuator.

11.3.1 Execution

The person following behavior can be executed in one of two ways:

1. The behavior can be run as a sub-behavior of the CMAssist top-level behavior `demo.py`. This is the preferred method of execution as verbal commands to the robot can then be given. To start the behavior issue a follow command like "Follow me." It will also be activated when training a task by demonstration ("Let me show you what to do when I say ..."). To stop the behavior, simply say stop.
2. The behavior can also be run from the command prompt by typing `python2.4 follow_person.py`. The behavior then immediately begins to search for someone about a meter away that has both leg and vision data in agreement. To stop the behavior, you must use `Ctrl+c`.

11.3.2 Verbal feedback

The person following behavior can understand the following utterances as feedback from the human to correct its behavior:

- “You’re/You are doing the wrong thing.” - The behavior marks the track it was following as bad and looks for a new one.
- “You’re/You are following the wrong person.” - Same as above.
- “You’re/You are going the wrong way.” - Same as above.
- “I’m/I am over here.” - Same as above.
- “I’m/I am (on — to) (your — the) left.” - The behavior marks the track it was following as bad, along with all the other tracks to its right, and looks for a new one.
- “I’m/I am (on — to) (your — the) right.” - The behavior marks the track it was following as bad, along with all the other tracks to its left, and looks for a new one.